

Block 2, Part 1: Architectures

Steven Self

1 Introduction

In this first part of Block 2, we will provide an overview of the high-level design of web applications and services. Design at this level is often termed the 'architecture' of a system. Architecture specifies the different components that make up a system, their functionality and how the different parts interact or communicate.

For this part of Block 2

For Activity 3 in this part of Block 2 you will be looking at the 'What is Cloud Computing?' (Amazon Web Services, 2017) website. You might wish to bookmark the website now.

2 Learning outcomes

On successfully completing this part's study, you should be able to:

- outline the development of application architecture leading toward a more modular approach
- sketch out the components that would be used in a range of approaches to application architecture
- outline a range of benefits and potential problems associated with a specific approach to application architecture
- describe the basic differences between and properties of thick and thin application clients
- contrast developments in architecture (SOA, Cloud) against more traditional tiered approaches
- outline how a high-level architecture can be refined to a 'design' and 'implementation', first in terms of components and protocols, and secondly, how it can be realised in terms of items such as products and programming languages.

3 Architecture

In Block 1, you were introduced to web architecture through the client-server model.

Traditionally, 'architecture' is a term normally associated with the design, style and structure of buildings. In the context of the architecture of web applications we will focus on structure, both in the terms of their physical components – the 'system architecture', and software (the programs making up an application) – the 'software architecture'.

Let's first define what we mean by 'software architecture' by considering the following interpretations.

According to Shaw and Garlan in *Software Architecture: Perspectives on an Emerging Discipline*, (1996):

The architecture of a software system defines that system in terms of computational components and interactions among those components.

The *Microsoft Application Architecture Guide* (2009) cites Kruchten *et al*'s definition of architecture based on the work by Shaw and Garlan:

Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behaviour as specified in collaboration among those elements; composition of these structural and behavioural elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, tradeoffs and aesthetic concerns.

In 'Patterns of Enterprise Application Architecture', Fowler (2002) states:

'Architecture' is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change.

In *Pattern-Oriented Software Architecture, Volume 4, A Pattern Language for Distributed Computing*, Buschmann *et al.* (2007) cite the following quote from Booch (2007, p. 214) which emphasises that design and architecture cannot be divorced from one another:

All architecture is design but not all design is architecture. Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change.

In *Software architecture for developers*, Brown (2012) describes architecture more widely:

So, we can see that the word ‘architecture’ means many different things to many different people and there are many different definitions floating around the internet. For me, architecture comprises four things:

- Structure – the building blocks (components) and how they relate to and/or interact with one another.
- Foundations – a stable basis on which to build something.
- Infrastructure services – the essential services that are an integral part of whatever is being built. With a building, this might be power, water, cooling, etc. With software, this might be security, configuration, error handling, etc.
- Vision – it is crucial that you understand what it is you are building and how that process will be undertaken. Vision can take the form of blueprints, guidelines, protocols, leadership, etc.

There are quite a few other descriptions of architecture which, like those from Shaw and Garlan and Microsoft, focus on components and have similar wording to Achimugu et al. (2010):

The foundation for any software system is its architecture. Software architecture is a view of the system that includes the system’s major components, the behaviour of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the overall system’s goal. Every efficient software system arises as a result of sound architectural basement. This requires the use of good architecture engineering practices and methods.

In summary, a ‘software architecture’ relates to the arrangement of software components and the interactions between these components. It also relates to the decisions that are taken about the design and composition of the overall system, how it is broken down into components and how these components interact. This type of structuring reflects high-level decisions that are difficult to change later because they determine software-level decisions. It is thus difficult to change the architecture of a system without starting the design from the beginning.

Activity 1 Defining ‘system architecture’

 Allow 15 minutes for this activity

Produce your own definition in answer to the question ‘What is system architecture?’ To do this, consider the definitions of a software architecture given above, and briefly research definitions for ‘system architecture’ and ‘design’ for yourself.

Reveal answer

‘Client-server’ and ‘multi-tier’ approaches represent mainstream common ‘tried and tested’ approaches, whereas ‘network type’ approaches represent more recent developments (in terms of how computers are connected and data are passed and processed between them) that many organisations are using, such developments have been shown to have additional benefits over the established approaches.

By looking at the development of architecture it is possible to understand the potential benefits of adopting one approach to application design over another, and to understand why a specific architecture itself imparts specific qualities that can lead to a better (or worse) solution.

4 Client-server and multi-tier architecture

4.1 Two-tier architecture

You saw in Block 1 that a **client-server architecture** divides an application into two distinct parts; 'client' and 'server'. Such an application is implemented on a computer network. The server part of that architecture provides the central functionality: that is, any number of clients can connect to the server and request that it performs a task. Assuming the requests are accepted, the server accepts the request, processes it and returns any results as a response to the client, as appropriate.

Consider an online bookstore as an example. The application allows a user to search and look at the details of a large range of books, and then to order a book. The application software provides an interface and a means of selecting or finding a book's details, as well as displaying book information and allowing a book order to be generated.

In the example, the bookstore application could take the form of a single 'chunk' of software downloaded from the web. However, if the software is one monolithic item, then every time anything is changed or updated, the entire application has to be redistributed again. Obviously this would not work well in this example because it is likely that a catalogue of books will change regularly. An improvement might be to split the application into two distinct parts. One part, the client, can provide the interface for users and be distributed to them. The second part can be kept and run from the company's own server machine (Figure 1). The client application can display information and be used to pass information to the server for searching, such as the title of a book. This client application, or software, is quite commonly called the 'presentation' layer or tier.



Figure 1 Simple client–server application

[Maximise](#)

In this client-server model, many clients can connect to the server application and request information about books. The server has to process the request and send the response to the client that originated the request and not to any other client. As long as the network is working well and the server can keep up with responding to all the requests it receives, such a 'split' application will provide much the same level of service as the monolithic version. This simple client–server architecture is also commonly called 'two-tier architecture'.

In this example, the catalogue of book information can be held centrally on the server and then be easily updated. This allows other 'centralised' information to be maintained and sent to clients, such as the stock level of each book. Users of the client will find it much simpler and smaller to work with than the complete application. At the same time, the company will have better control of the server application itself, for example, usage. A common client used to access applications is a web browser that accesses server applications (such

as applications on websites) using HTTP. The use of a web browser as the client end of an application is interesting because, for most applications, the browser is often provided by a third party. This means that application builders must rely on agreed standards for the behaviour of the client component.

There are also different distributions of functionality across a two-tier architecture. For instance, suppose you have a client that accesses your bank account online. If that client is a web browser, it can be used to request and display your accounts' statements from the bank's server. The information you may obtain is restricted to the pre-defined views of the information provided by the server. So, while the server's information might include your account balance, if you want to find out the total payments in and out over the last week or year, then you may still have to calculate it yourself, based on the figures the server provides. Alternatively, you might access your bank server over the internet using another, 'more intelligent' client. This client might include a facility to extract figures from your bank statements and to perform whatever calculations you require. Such a client might also create bar or pie charts that display your income and expenditure across different categories that you define.

The web browser client in the example of the online bank simply displays the information that the server provides. The 'more intelligent' client enables you to take the information the server provides and to manipulate and display it in various ways according to your own personal needs. Many banks provide their own clients to improve their service. The web browser with little functionality of its own is often termed a **thin client** and the 'more intelligent' client, such as a mobile app, is usually termed a **thick (or 'thicker') client**.

You have seen that a two-tier approach seems to have some advantages, at least for applications that operate over networks. The client that is distributed to users may change, while the server part can be seen as a centralised component that maintains dynamic, global data in a consistent and secure way for the organisation and for users to access and use. If a third party component, such as a web browser, provides the functionality required to support the application, then it can be adopted as part of the solution, with a significant saving of development effort. There is a potential disadvantage from splitting the application across a network in that data has to be transmitted over a connection that at times may be slow or unreliable connection.

There are other, slightly less obvious advantages in breaking the application into components in this way. We shall look at some of these in more detail shortly, but let's look at one advantage now. This is an advantage from which you will have benefited if you have ever decided to change your web browser. You might have changed from Internet Explorer to Mozilla Firefox, for example, just as a personal preference. When you did this you changed the client component of all the online applications you use. This is only possible because web browsers are based largely on common standards and because they are not realised as an intrinsic or built-in part of any of the applications you use; in other words, the client is loosely coupled to the server application.

4.2 Multi-tier architecture

The approach of splitting an application into tiers can be taken further. When we talk about multi-tier architecture, both the client and the server parts can be further subdivided if this is appropriate for the application. The client, for example, may be responsible both for some processing of data received and for the presentation of information. In our example of a bookshop, we may need to calculate the total cost of an order and display sample content of books. These two functions might be separated into two different tiers at the client end.

The server software might include one or more data stores (for instance, in the form of a database system). In the example of a bookshop, one data store might include the images, cost and reviews of a book; another might be used to store more dynamic information, such as the current stock levels or delivery time for each book. Dividing data up in this way would, for example, allow occasional backups of the more static data with more frequent backups of the dynamic data.

So, an architecture used over the internet might look in outline like that shown in Figure 2. Here we have added two more tiers on the server side: a data tier and another tier that handles interactions with the data tier, such as retrieving requested information or validating data that is put into the data stores. This tier, between the data tier and the **web server** application, is sometimes termed the 'middle tier' or **middleware**. An application using middleware to handle data requests between a user and a database is said to employ **multi-tier architecture**.

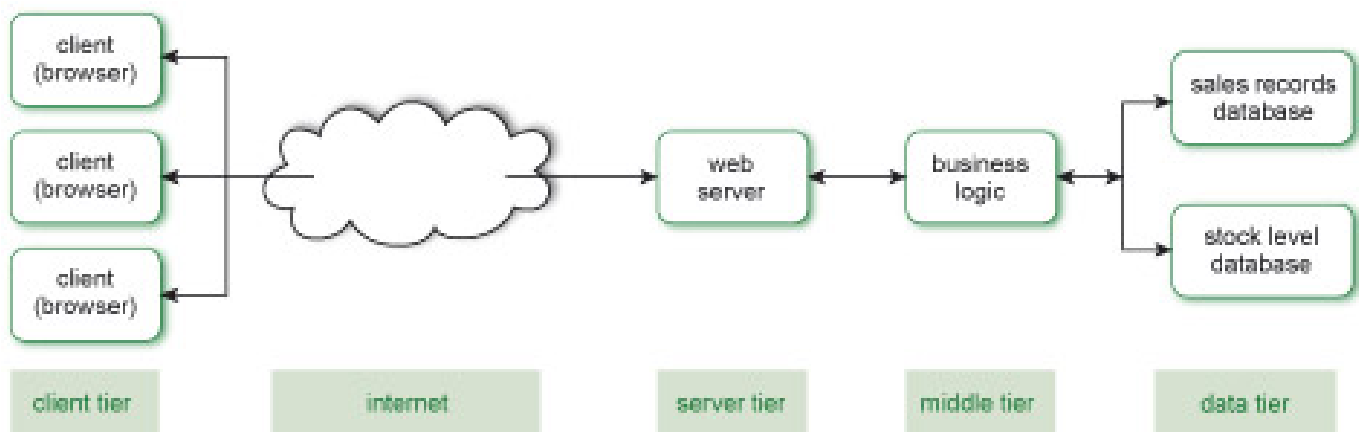


Figure 2 Example of multi-tier architecture

Maximise

Quite commonly, 'multi-tier architecture' refers to what should more specifically be called three-tier architecture (client, server and data tiers). More tiers than this, however, can be used (as illustrated in Figure 2) and so the term 'N-tier architecture' is used generally to mean any architecture that has more than two tiers.

As in the two-tier approach, there are advantages to breaking down an application into multiple tiers. Each tier can be changed more easily as it is less dependent on the precise details of the other tiers with which it interacts. This again depends on how careful we are to adopt a more modular approach that ensures the tiers or components are truly loosely coupled. Simply breaking the application into chunks doesn't guarantee this; we also need to adopt suitable standards, and specify precise and limited interactions between the tiers. N-tier architecture has almost become an all-pervasive approach, and is certainly very mainstream. An advantage to N-tier architecture, therefore, is that many more components besides web browsers are now available that can realistically be interchanged without prohibitive effort, including databases, web servers and some middleware components.

5 Network and distributed architectures

5.1 Service-oriented architecture (SOA)

As our description of architecture has moved from monolithic applications to client-server and then to N-tier, so the application has been broken down into more and more parts. This trend has been extended in a modern approach called service-oriented architecture (SOA). SOA is based around the idea of breaking down an

application into a set of much smaller tasks that can be performed by small independent pieces of software, software components, each performing a discrete task commonly called a service.

SOA is the architectural solution for integrating diverse systems by providing an architectural style that promotes loose coupling and reuse. The software components provide services to other components via a communications protocol, typically over a network. The party offering the service is known as a *service provider* (a server), and the party invoking the service a *service consumer* (a client).

A service is some functionality, typically a business process, which is packaged as a reusable software component that is:

- 'well-defined' – a software component with a clearly specified interface and outcome
- 'self-contained' – the implementation of the service is complete and independent of any product, vendor or technology
- a 'black-box' – the implementation of the service is hidden (encapsulated) from the service consumer.

Examples of services might be:

- **Currency conversion:** A service that converts a sum of money from one currency to another might be used in a wide range of circumstances. It might be used, for example, to convert all the prices on a website store to a customer's local currency.
- **Check customer credit:** A service that provides the credit rating for a given customer.
- **Provide weather data:** A service that provides selected weather data for a given geographical area and time period.
- **Data storage:** A service that allows data to be stored and later retrieved, perhaps with a range of capacities, costs and timescales on offer.

SOA principles

SOA is governed by a number of design principles (Erl, 2007):

Interoperability: services implemented in different systems, locations or even business domains, and using diverse technologies – ranging from programming languages to platforms – need to work together to allow diverse consumers and providers to communicate. For this to happen effectively, services must rely on internationally agreed communication standards. Many of the current standards were developed for communication over the web – although SOA is in fact technology-agnostic, so that its principles apply regardless of which communication standard is actually used.

Location transparency: consumers should be able to make use of a service without necessarily knowing where the service is located, that is, on which machine or even in which business organisation. Service consumer and provider are only loosely bound together, with the consumer having no knowledge of the service implementation or its platform, and consumer and provider communicating solely through messages.

Discoverability: the consumer must be able to find out about relevant services, which usually involves access to appropriate metadata about services. Discovery is often done through registries: a **service registry** will contain metadata and references to services. The actual services are often contained in inventories: a **service inventory** is collection of complementary services within a boundary of the provider, an enterprise or even a meaningful segment of an enterprise. We distinguish between run-time and design-time discovery. In **run-time** (or **dynamic**) **discovery**, the consumer first queries the registry for a service that matches the consumer's criteria; if such a service exists, the registry provides the consumer with the location of the service provider; the consumer can then bind dynamically with the service provider and invoke the service. This model of service collaboration is known as the '**find, bind and invoke**' cycle. While this model is of historical and theoretical significance, it has yet to become mainstream in practice despite much research and development effort in both academia and industry. This is due to the intrinsic difficulty of specifying semantic information which may allow one to reliably predict the interaction of a potentially unlimited number of diverse services. Instead, current discovery mechanisms support primarily **design-time discovery** – that is, they're aimed at developers who are trying to find out about relevant services in the planning stage of a SOA application development.

Loose coupling and encapsulation: coupling between service providers and consumers should be low, and confined to reliance to service public interfaces, with an understanding that such interfaces should disclose as little as possible of the underlying encapsulated implementation details. The interfaces should be based on standard communication protocols, rather than proprietary ones, with open standards recommended to foster the highest degree of interoperability. Also, asynchronous communication should whenever possible be preferred over synchronous communication, as the former affords further decoupling by removing – for instance – stringent availability and performance constraints on the provider.

Abstraction: both the encapsulated implementation, the implementation technology and the physical location of services should be invisible to consumers, which should be reliant solely on public interfaces, service contract descriptions and SOA infrastructures to locate and invoke required services.

Autonomy: the more autonomous a service is, the more control it will have over its own implementation and run-time environment, and so the greater will be its flexibility and potential for evolution. Both implementation and run-time environment can be modified without affecting consumers.


Statelessness: excessive state information can compromise the availability of a service and limit its scalability; hence, services should remain stateless as far as is realistic to allow them to do their work.

Standardised interfaces and contracts: to support the service collaboration model and automate service interaction, services need to be described in a similar way using commonly understood standards. In particular, a service description may include descriptions of the service technical interface (that is, the operations which can be invoked and their parameters), the quality of service provided, and a service-level agreement which details both service characteristics (e.g. response time or availability) and cost to the consumer of invocation. Additional metadata may cover a range of information, for instance including user satisfaction rating or future development plans.

Reusability: services must be designed with reuse in mind, so generality of the service in terms of potential reuse across projects and systems is an important design principle. This impacts on both the granularity of the service (the service should strive to support generic business processes and tasks and close alignment with the business) and on the choice of standards and implementation technology, which should foster the highest possible degree of interoperability.

Dynamic reconfiguration: service-based systems should be configurable and reconfigurable dynamically by discovering and incorporating existing services, but while maintaining coherence and integrity. This should be supported by metadata, which underlie the discovery of existing services, coupled with contract agreement descriptors.

Activity 2 SOA

 Allow 15 minutes for this activity

Given what you've just learnt about services and SOA design principles, which do you think are the advantages of developing business applications based on SOA?

Reveal answer

Web services

When a service is made available over the internet, it is then usually termed a web service, which describes the service that a client can access from a server over the internet, utilising web protocols and standards to enable the exchange of data between them.

The main protocols and standards employed are SOAP (Simple Object Access Protocol), REST (Representational State Transfer), XML (eXtensible Markup Language) and JSON (JavaScript Object Notation) based.

Discussion of these protocols and standards in any depth is outside the scope of this module (other modules do explore these in detail). For us, the main point to make is that web services based on the SOA design principles described will ensure that web applications written in various programming languages and run on a variety of platforms, and can use web services to exchange their data across computer networks in a manner similar to inter-process communication on a single computer.

5.2 Cloud architecture

SOA is well placed to take advantage of a recent development known as 'Cloud technology' based on what is often termed 'the Cloud'. Cloud technology has changed how organisations use technology to provide computing services. There is, however, a lack of clarity as to what the term will really come to mean. Some definitions restrict the Cloud to mean that **virtual servers** are made available and used over the internet, but more generally the Cloud is seen as consisting of a wide range of different resources.

The National Institute of Standards and Technology (NIST) provides a wider definition of Cloud computing:

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

(NIST, 2011)

Cloud computing is the practice of delivering computing services – servers, storage, databases, networking, software, analytics and more – on-demand over the internet. It is a means of providing computing services as a utility to consumers in the same way as other utilities such as gas and electricity. Companies offering these computing services typically charge for cloud computing services based on usage.

You might take the view that whereas SOA provides services on a network, the Cloud extends the principle to other resources such as processing power and storage facilities and is not just limited to the provision of services. There are, of course, quite a lot of details that are not covered here. How resources are charged for, made secure, 'cleaned' after use, etc. are just some of the aspects that need to be considered in the Cloud.

Activity 3 Benefits and potential problems of using Cloud architecture

 Allow 30 minutes for this activity

Given the NIST view of Cloud computing, now take a look at the Amazon Web Services (AWS) website
What is Cloud Computing?

Considering the video presentation and supporting text, what do you see as the benefits and potential problems of using Cloud architecture?

Are you, or the organisation that you work for, using Cloud services?

Post your comments to the Block 2 Forum.

Cloud computing is itself a model for enabling convenient, on-demand network access to a shared pool of configurable hardware, software and network computing resources and services that can be rapidly provisioned and released with minimal management effort or service provider interaction.

NIST (2011) defines five essential characteristics:

- **On-demand self-service.** A consumer can automatically and unilaterally provision computing capabilities such as server time and network storage as needed, without requiring human interaction with each service provider.
- **Broad network access.** Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g. mobile phones, tablets, laptops and workstations).
- **Resource pooling.** The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth.
- **Rapid elasticity.** Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities

available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

- **Measured service.** Cloud systems automatically control and optimise resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g. storage, processing, bandwidth, active user accounts). Resource usage can be monitored, controlled, audited and reported, providing transparency for both the provider and consumer of the utilised service.

NIST (2011) defines three service models:

- Software as a Service (SaaS) – provides applications designed for end-users, delivered over the web.
- Platform as a Service (PaaS) – provides tools and services for developing and deploying applications.
- Infrastructure as a Service (IaaS) – provides servers, storage, network services and virtual machines.

As these service models offer increasing levels of abstraction they are often depicted as a stack of service layers – infrastructure, platform and software-as-a-service layers – though they may not be physically implemented by a tiered architecture.

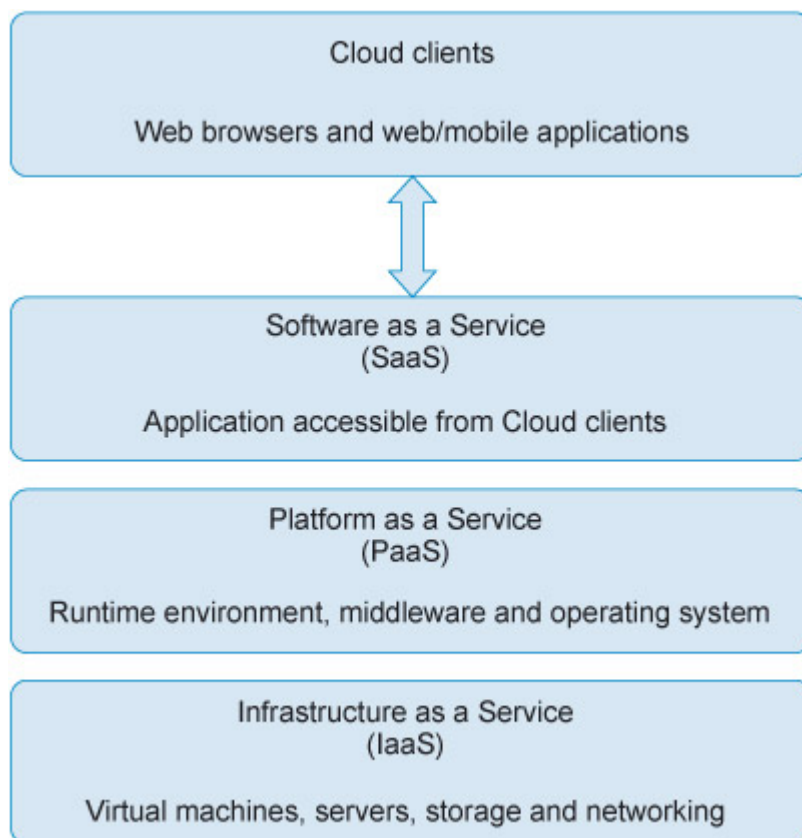


Figure 3 Cloud computing depicted as a stack of service providing layers

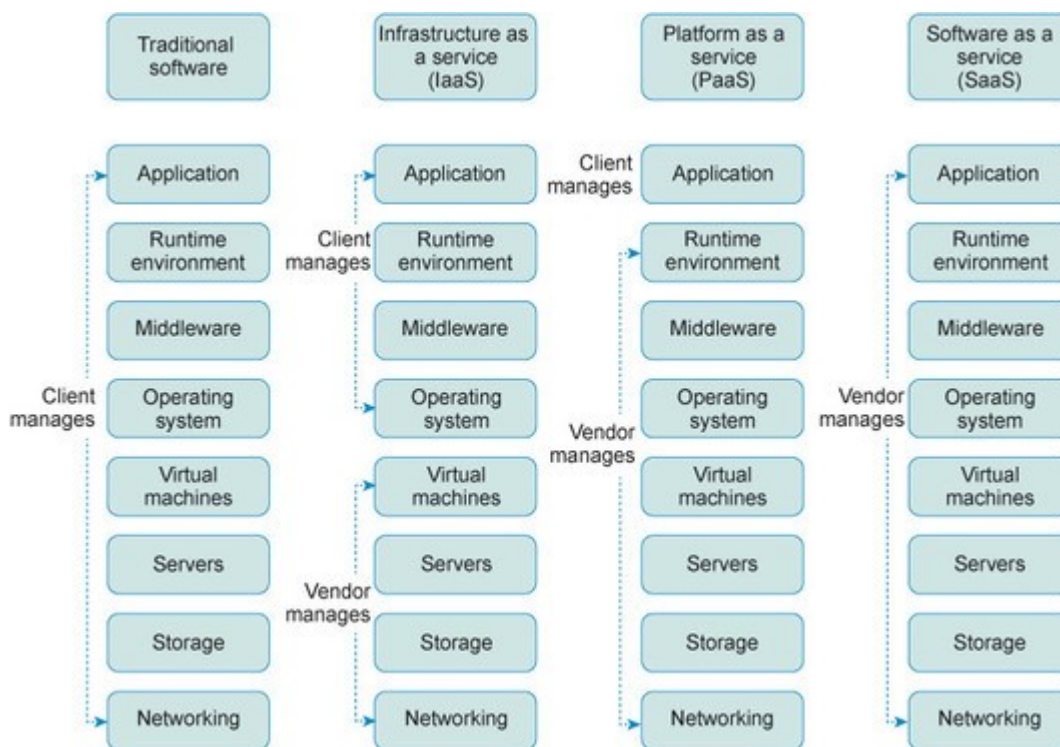


Figure 4 Cloud computing service models depicted as a stack of component layers

Software as a Service (SaaS)

NIST (2011) defines SaaS as:

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g. web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.

With SaaS, a service provider licenses an application to customers either as a service on demand, through a subscription, as a "pay-as-you-go" model, or (increasingly) at no charge when there is opportunity to generate revenue from streams other than from the user, such as from advertisements. SaaS is a rapidly growing market and it is expected that SaaS will soon become commonplace within most organizations.

Advantages of SaaS include:

- the provision of software that would be normally installed on users' computers (e.g. Microsoft Office) as web applications
- the entire application and hosting infrastructure is the responsibility of the service provider (Figure 4)
- users are not required to implement software upgrades and patches
- the provider can provide scalable web applications using a multi-tiered architecture, implemented on a high-performance infrastructure.

With SaaS a significant amount of the processing occurs on service providers' computers and not on the service consumers' computers.

There are numerous examples of SaaS available, many based on subscription fees like Netflix and DropBox. Organisations can also use the SaaS model to provide email, word processing and other 'office' tools for their employees. A recent trend is for application developers to convert from stand-alone applications to a web-based SaaS delivery model. For example, Microsoft Office 365 is a group of software and service subscriptions which together provide access to Microsoft Office applications and Cloud storage services to subscribers. The SaaS model gives the developer greater control since they deploy to a known environment in the Cloud rather than distributing thousands of copies to individual users.

Platform as a Service (PaaS)

NIST (2011) defines PaaS as:

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

PaaS provides an infrastructure for developing web applications. PaaS is analogous to SaaS except that, rather than being software delivered over the web, it is a platform for the creation of software, delivered over the web.

Features of PaaS include:

- Services to develop, test, deploy, host and maintain web applications in the same integrated development environment.
- Web-based user interface creation tools help to create, modify, test and deploy GUIs.
- Support for development team collaboration – a multi-user environment to allow several users to work on the same application concurrently.
- Integration with web services and databases.

PaaS is especially useful in any situation where several developers are working on the same development project. It also is useful where developers wish to automate testing and deployment services.

Infrastructure as a Service (IaaS)

NIST (2011) defines IaaS as:

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g. host firewalls).

IaaS delivers a cloud computing infrastructure – servers, storage, network services and virtual machines – as services. Rather than purchasing these infrastructure components, clients instead purchase access to these components as services.

Using IaaS is particularly suitable for situations including:

- new organisations without the capital to invest in hardware
- organisations that are growing rapidly but purchasing additional hardware would be prohibitive
- speculative development of new lines of business without the need to invest in infrastructure components.

Examples of IaaS providers include Amazon Elastic Compute Cloud (EC2) and Google Compute Engine.

Amazon Elastic Compute Cloud (EC2) forms a central part of Amazon's Cloud-computing platform, Amazon Web Services (AWS), which supports both Amazon's retail services and Cloud services.

Google Compute Engine is the IaaS component of Google Cloud Platform which is built on the global infrastructure that runs Google's search engine, Gmail, YouTube and other services.

Cloud computing provides three service models, these are IaaS as the underlying infrastructure, PaaS as a web application development environment and SaaS which replaces stand-alone applications with web applications. These three service models are key to understanding how the Cloud has evolved, so it is important that you appreciate the key differences between them.

Cloud computing brings with it a number of key benefits, as well as risks, that should be carefully examined by any organisation looking to move to Cloud computing. It is important for organisations to understand the different aspects of Cloud computing and to assess their own requirements before deciding which service models are appropriate for their unique needs. Cloud computing is a rapidly accelerating revolution within IT and is likely to become the default method of IT service delivery in the future.

You have now completed the rapid tour of architecture in Block 2. Before we move on to investigate specific parts of a typical layered web architecture in more detail, we want you to look quickly at how to bridge the gap between the high-level architectural outlines given earlier and a working set of components that can be used to realise an architecture.

6 Realising an architecture

Now that you have an overview of architecture and some common approaches as background, we will move on to take a quick look at some software products that provide one or more components of a web application.

We have missed out a step in moving from architecture to looking at products, which is the issue of 'design'. If we briefly return to Figure 2, where multi-tier architecture is depicted, it can be seen that in the data tier there is an aspect of 'design' in that there are some choices that can be made. In terms of an architecture, we might specify that there is a data tier and it will, for the application in question, include information about sales and stock levels. A design decision has been made to put the information into two separate databases.

The data for sales and stock level applications might be stored in a single database, or it might be stored in separate databases. If data is stored in separate databases, then these databases can also easily reside on different machines if required. These decisions depend on requirements such as access to data, security, backup schedules, etc. So, for example, if the data for one application has to have very restricted access, such as credit card details, then it may well be easier to keep that data in a single database on a separate computer that has very restricted access.

By looking at components and how they might be used in an architecture we are moving from the high level, more abstract architecture toward more concrete implementation aspects in a top-down fashion (illustrated in Figure 5). It would be quite rare to have the luxury of taking this approach freely in a real setting because there would be other constraints on the choices available. Constraints would include the need to integrate with existing systems, or to adhere to some organisational policy in selecting components and how they can be used.

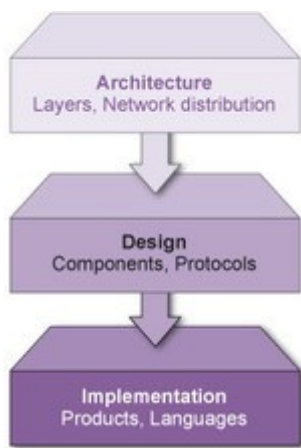


Figure 5 Top-down refinement of architecture to implementation

In some instances, design components are intended to be used within a specific architectural approach and can't be used with others.

Here we have focused on the architecture of applications from a technical viewpoint and at quite a high conceptual level. There are other types of architecture and other factors, such as business factors, that will have an influence on the approach to development of applications in an organisation.

Next, we are going to visit the bottom level of the top-down refinement in Figure 5 by taking the architecture in Figure 2 and suggesting one example of a set of products that could be used to realise much of this in the development of an application.

7 A few products

There is a large range of products available to us when designing web architectures and in the following section we'll look at a few mainstream examples. As a focus, we will revisit the 'multi-tier architecture' (Figure 2) and suggest some 'products' and solutions we might use to realise one instance or version of this architecture.

7.1 Client tier

You probably know quite a few potential client browsers that might be used (see StatCounter for statistics on browser use or What's the best source for browser usage stats? for a discussion of different sources). We need, however, a browser that is fully HTML5 and CSS3-compliant, such as Firefox or Google Chrome, to ensure that the HTML and CSS code you saw in Block 1 renders as expected.

7.2 Server tier

For the server tier we will use a web server. We have several choices, the most popular being Apache, NGINX and Microsoft's IIS (see https://w3techs.com/technologies/overview/web_server/all for statistics on web server use). In our case, we'll choose Apache, an HTTP server, as we are very familiar with this and it is freely available as open source software. There are other, more commercial versions available if we have a requirement for support, for example, IBM HTTP Server or Oracle WebLogic.

7.3 Middle tier

For the middle tier, which supports the business logic for applications, we need to pick a programming language in which to express that logic. A number of programs need to be written to perform the operations that carry out this logic. So, for example, based on Figure 2 we would expect there to be a program that updates the level of stock in a database when sales of an item are made. Depending on our choice of language we may or may not need some other server software.

Our server tier choice 'Apache' is extended with 'modules' that can run languages such as PHP (a server-side scripting language for web applications) so, if we were to choose that language, we could code in PHP and run the business logic programs on the same server hosting our HTML web pages. If we prefer to program in Java, however, we could add an application server to run Java programs. Our choice might then be Tomcat, a web server that interacts with Java programs. This application server is also an 'Apache' project and can communicate with the Apache web server so they can be expected to work well together.

7.4 Data tier

For the data tier we also have a range of choices. In this block we are going to choose MariaDB as this is another freely available mainstream product. MariaDB is a database management system (DBMS) allowing any number of actual databases to be created for holding and managing data, so we can easily have both the sales records and stock levels databases shown in Figure 2. The overall picture of the architecture using our selection in 'product' terms now looks as shown in Figure 6.

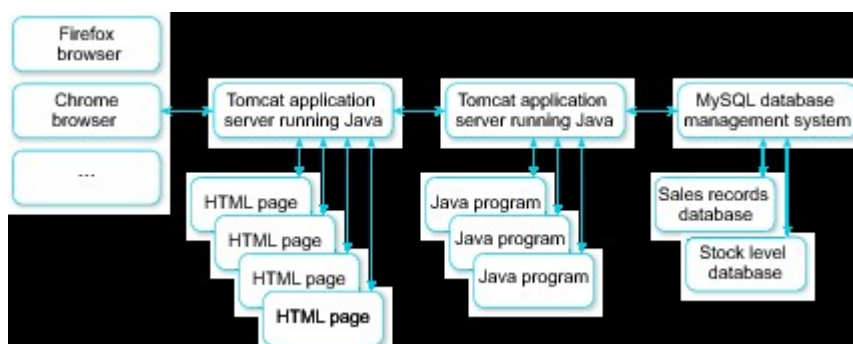


Figure 6 Outline of multi-tier architecture realisation

We may have selected our preferred web server, but there are a lot of other decisions we need to make in order to realise an application. In terms of serving our data, what web pages will the server host and what will they do? How does communication work with the middle tier if this is hosted on another server? How many web servers do we need?

If we were putting forward a more reasonable argument for the choices above, we would have to cover quite a bit more than these aspects (popular, open source, etc.). Our personal preferences alone would also not be valid as arguments. In considering the technologies we would expect to cover license terms, costs in detail, skill requirements, interoperability, performance, flexibility, future development, documentation, support and more. One valid argument for our choice of both a web server for HTML pages and an application server to run Java programs (Figure 6) is that the web server can pass requests that require Java processing onto the application server, which is specialised and efficient in handling these, so that it is not burdened with this work and can therefore respond very rapidly to requests for static web pages.

Activity 4 Proposing software products

 Allow 15 minutes for this activity

We have listed some of the central considerations that should be made when evaluating software. Produce your own list that is as exhaustive as possible, making sure you are comfortable with the meaning of each term.

Post your ideas and any queries you may have to the Block 2 Forum.

8 Reflecting on architecture

At this point you might wish to consider why we looked at 'conceptual architecture' when what we really needed to produce was a practical solution composed of real items that can be downloaded and used. In terms of architecture, the answer may lie in a broader view of a range of considerations.

1. First, we now have a common language and understanding. In future you should be able to draw or understand a diagram using terminology such as 'client', 'data tier', 'middle tier', 'business logic', and to be comfortable discussing such a diagram with colleagues. So the architecture is a vehicle for common understanding and discussion of the overall implementation.
2. The conceptual architecture, whichever approach is taken, is not a commitment to any product or set of products and allows a solution, or the approach to a solution, to be planned and justified in isolation without the clutter of details.
3. Once the architecture is settled, it then serves as a design against which to plot the progress of the implementation. The language can be 'Do we have a preferred product to support the business logic?'
4. Later, after implementation, the conceptual architecture serves as the starting point for understanding what has been implemented, but not how it has been implemented.

Activity 5 To architect or not?

 Allow 30 minutes for this activity

Consider the usefulness of design at the architectural level; is it more valuable if working on a large project or on a small 'one-or two-person scale' development project?

Post your ideas about the usefulness and benefits of designing architecture in the Block 2 Forum.

9 Summary

We have introduced a range of different application architectures and briefly explored some of their properties. We have also briefly investigated an example of realising a multi-tier architecture in software terms using a range of 'products' and a single programming language.

We will next focus first on client-side aspects, before moving on to server-side programming, and finally on the data tier. There will be many important ideas and concepts to understand.

The final 'achievement' of the practical work over these parts will be that you will see how to create the client, server and database components of a small application very similar to that in Figure 2 (but with a very simple middle tier), but rather we shall choose PHP over Java, and run the business logic on the same Apache server where web pages are hosted. Later in the block, you will install XAMPP to achieve this.

References

Achimugu, P., Babajide, A., Oluwaranti, A., Gambo, I. and Oluwagbemi, O. (2010) 'Software architecture and methodology as a tool for efficient software engineering process: a critical appraisal', *Journal of Software Engineering & Applications*, 3, pp. 933–38.

Brown, S. (2012) *Software Architecture for Developers*. Available at <http://leanpub.com/software-architecture-for-developers> (Accessed: 5 June 2023).

Booch, in Buschmann, F., Henney, K., Schmidt, D.C. (2007) *Pattern-Oriented Software Architecture, Volume 4, A Pattern Language for Distributed Computing*, Chichester, Wiley.

Erl, T. (2007) *SOA: Principles of Service Design*, Prentice Hall.

Fowler, M. (2002) *Patterns of Enterprise Application Architecture*, Boston, Addison-Wesley Longman Publishing.

Kruchten et al., Microsoft (2009) 'Chapter 1: What is Software Architecture?' Available at: <https://msdn.microsoft.com/en-gb/library/ee658098.aspx> (Accessed: 5 June 2023).

NIST (2011) '*The NIST Definition of Cloud Computing*', National Institute of Standards and Technology. Available at <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf> (Accessed: 5 June 2023).

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Further reading

A range of links are given below if you wish to learn more about some of the technologies or products mentioned in this part of the block. You are not required to read any of these at this time but we will be encountering most of these areas later in the block when you will be told which of them are necessary or useful for completing the block's practical elements.

Amazon Elastic Compute Cloud (EC2) (2017) available at: <https://aws.amazon.com/ec2/> (Accessed: 5 June 2023).

Apache (2017) available at <http://httpd.apache.org/> (Accessed: 5 June 2023).

Google Compute Engine (n.d.) available at: <https://cloud.google.com/compute/> (Accessed 5 June 2023).

IBM HTTP Server (n.d.) available at: <http://www-01.ibm.com/software/webervers/httpservers/#> (Accessed 5 June 2023).

IIS (no date.) Internet Information Services. Available at: <http://www.microsoft.com/web/platform/server.aspx> (Accessed: 5 June 2023).

JSON (no date) '*Introducing JSON*'. Available at <http://www.json.org/> (Accessed 5 June 2023).

MariaDB (2017) available at <https://mariadb.org/> (Accessed: 5 June 2023).

NGINX (n.d.) available at: <http://nginx.org/en/> (Accessed 5 June 2023).

Oracle WebLogic Server (no date) available at Oracle WebLogic Server.(Accessed: 5 June 2023).

PHP (2017) available at: <http://www.php.net/> (Accessed: 5 June 2023).

REST (2015) *Representational State Transfer*. Available at <http://www.ibm.com/developerworks/webservices/library/ws-restful/> (Accessed: 5 June 2023).

Service-oriented architecture (SOA) standards (2012) *IBM DeveloperWorks* [online]. Available at: <https://www.ibm.com/developerworks/library/ws-soa-standards/> (Accessed 5 June 2023).

SOAP (2004) *Simple Object Access Protocol*. Available at: <http://www.w3.org/TR/soap/> (Accessed 5 June 2023).

Tomcat (2017) available at: <http://tomcat.apache.org/> (Accessed: 5 June 2023).

Where next?

In Block 2 Part 2, we will examine client-side architecture and look at how we include JavaScript in web pages.

